Beyond the Prototype: The Design Evolution of a Deployed AI System

Cheryl Martin and Debra Schreckenghost

NASA Johnson Space Center, TRACLabs
1012 Hercules, Houston TX, 77058, USA
cmartin@traclabs.com, ghost@ieee.org

Abstract

This paper describes the evolution, from prototype to deployment, of the software engineering approach for a personal assistant agent-based system. This discussion is presented as a case study, relating experiences and lessons learned from our work with the Distributed Collaboration and Interaction (DCI) environment. Our development of this system is based on the spiral software engineering methodology, which incorporates iterative cycles of improved design, implementation, and evaluation. We first describe the techniques we used to bootstrap the implementation of this large, distributed AI system, and then we describe how we accommodated more advanced requirements as the system matured.

Introduction

During our previous experiences with deployed intelligent control agents for NASA advanced life support systems (Schreckenghost et al., 2002), we discovered many unaddressed needs for human interaction with control agents. These control agents operate continuously over months to years to monitor and perform process control for regenerative life support systems, which recover usable water or air from the waste products created by biological systems over time. To address these interaction needs, we developed the Distributed Collaboration and Interaction (DCI) system, which helps humans and autonomous control systems in a variety of ways to work together. This paper describes the software engineering aspects of our experiences, first, in designing and developing a prototype of the DCI system, and later, in adjusting the implementation to make the leap from initial prototype to a system ready to be applied under varying circumstances to meet different needs. The details in the paper focus primarily on the second stage of development.

Overall, we have implemented demonstrations of DCI in two domains and deployed the system in one of those domains. Specifically, we have demonstrated an application to support ground test engineers in monitoring life support systems and control agents as well as an application of DCI to support astronauts and ground personnel in mission support roles. The DCI system is currently deployed in the first domain, operating 24/7, to assist engineers interacting with an advanced Water Recovery System (WRS). A related paper from these proceedings (Schreckenghost et al., 2004) discusses other lessons learned during the deployment as well as user feedback and the management of the deployed DCI system.

DCI Overview

Figure 1 depicts representative elements of a DCI system. The entities with black backgrounds (the human, the WRS life support system and its control agent, and the multi-agent planner) participate in, but are not part of, the DCI environment. Our planner is a hierarchical task net (HTN) planner known as AP that is capable of automatically monitoring and updating its plans (Elsaesser and Sanborn, 1990). The Conversion Assistant for Planning (CAP) is software that augments an automated planner's ability to interface with human agents (e.g., when an organization uses planning to coordinate anomaly response activities between software and human agents). The Event Detection Assistant (EDA) monitors data produced by the life support system or its control agent and searches for patterns in this data that are of interest to the humans for such activities as anomaly analysis and failure impact assessment. The DCI approach uses intermediate *liaison agents*, one associated with each human, to mediate between that human and the

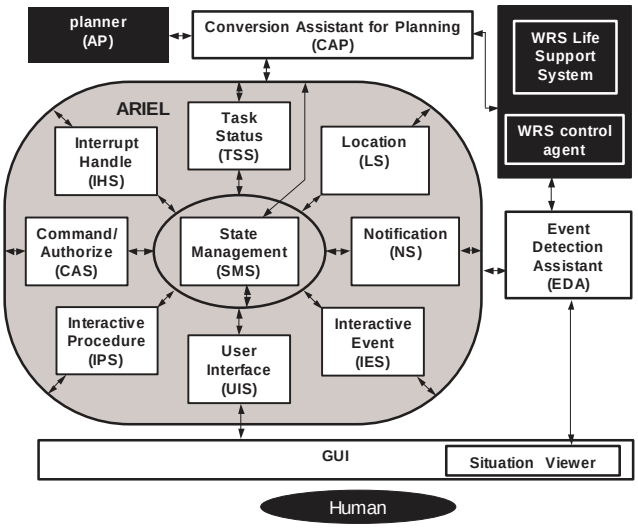

Figure 1. DCI System Architecture and Connectivity.

rest of a multi-agent system. A liaison agent is a type of personal assistant agent for its user. The liaison agents in DCI are called Attentive Remote Interaction and Execution Liaison (ARIEL) agents. Each agent offers a set of services, as shown in Figure 1. Full descriptions for the system and each of these services are given in (Martin et al., 2003). This paper refers to a subset of these services:

• *State Management Service (SMS)*: The SMS stores beliefs held by ARIEL and takes input from other services to create a coherent state model of the user. This model may include current activity, location, role, schedule, etc.
• *Notification Service (NS)*: The NS uses pattern-matching and specifications defined by a user or an organization to determine if an incoming notice or event is of interest to the user and, if so, how to inform her based on where she is and what she may be doing.
• *Location Service (LS)*: Tracks the user's location.
• *Task Status Service (TSS)*: Tracks the user's current activity and its status (acknowledged, initiated, etc.).
• *User Interface Service (UIS)*: Handles all Ariel communication with the user.
• *Command and Authorization Service* (CAS): The CAS supports the user in remotely issuing directives to physical systems that are otherwise controlled by mostly autonomous agents (e.g., the physical hardware underlying life support systems).

Initial DCI Prototype

As the first step in designing the DCI system and the Ariel agents, we identified what services the agents needed to provide as well as support software we needed for our application. That part of the design process is documented in (Schreckenghost et al., 2002), and the resulting system composition is shown in Figure 1. We faced several challenges in implementing a prototype for this design:

- *Distributed computing*. The DCI system is naturally distributed because it is required to support users who may be located remotely from the autonomous control system with which they are interacting. User GUIs are located remotely and location tracking and task tracking must also take place remotely.
- *Size*. The DCI system is large, with approximately 140 distributed server and client objects in the deployed system including Ariel service modules, DCI user GUIs, system access managers, etc.
- *Multiple platforms and programming languages.* In order to interface with existing software tools (i.e., a planner written in Lisp) and to take advantage of other language-specific capabilities (i.e., Java's cross-platform GUI support), we needed to support multiple programming languages. Also, based on the available resources at various locations, we needed to be able to execute on multiple platforms simultaneously.
- *Multiple developers.* A team of five software developers collaborated to implement the DCI prototype. Most of these developers were also working on other tasks, which meant they were not always available to immediately address problems in the developing software.

Balanced against these challenges, the design has the advantage of being very modular. Strong dependencies do exist among the components, for example, the Notification Service needs the Location Service to be able to tell where a user is, and whether he or she has online access, in order to determine the best means of notifying for that location. However, due to the modular design, these dependencies could be managed easily in most cases by succinct interface definitions for communication among the components. We reviewed the available technologies for implementing this modular design under these circumstances and determined that CORBA standards and tools (www.omg.org) met all requirements and offered the best technical advantages. CORBA supports a much wider variety of languages and platforms than other distributed computing technologies, it provides a great deal of transparent middleware and tool support for developers, and its specification is formal, well-documented, and well-architected. We chose not to use Java RMI for our cross-platform implementation because we needed to support Lisp and C++ programming languages. We chose not to use DCOM due to its limited or non-existent support for the platforms and programming languages we needed to use (particularly Lisp and Java under Linux).
At this point, we undertook the software engineering task of defining the interfaces among the components using CORBA Interface Definition Language (IDL). Some of these interface definitions were straightforward and served to get information from one service to another. For example, once the LS determines a current location for the user (originally modeled as a NASA crew member), it posts this information to the SMS:

```
void postLocation(in DCI::CrewLocation loc);
```

Using this approach and the CORBA tools for development, we reaped the classic advantages of the object-oriented methodology by encapsulating each service in a well-defined interface. This approach allowed individual developers to concentrate on their own internal AI algorithms such as pattern matching (in the NS) and belief revision (in the SMS, TSS, and LS), and it reduced dependencies on other components developed by other programmers. Although this approach is not uniquely useful for AI applications, it is especially important for integrating multiple AI technologies into a large deployed system. In essence, encapsulating the complexity of the AI algorithms themselves reduces the complexity of integration, which could otherwise become unmanageable. Further, this approach supports future upgrades to the AI technologies included in the Ariel services without requiring changes in the integration. Again, this benefit does not uniquely apply to AI technology, but it is particularly important because the AI algorithms are active areas of development and thus prone to change.

An interesting and extremely productive dynamic arose during integration testing. Because our CORBA-based approach supported full distribution, developers could debug remotely from one another on the *same* running system. Each developer could sit at his/her own computer, (often in different buildings, in the development environment of his or her choice) to run the integration tests and watch his/her piece of software running, make bug fixes locally, and run again. This was extremely efficient because it allowed each developer to simultaneously debug in his/her own development environment while seeing how his/her module performed in integration with other modules. This was particularly important for bootstrapping the system integration because many of the developers had never previously used CORBA, and there were many initial connectivity errors as they learned the technology.

The process of defining and implementing the static communication of information among the services, as just described above, was far more straightforward than the task of dynamically coordinating Ariel services. Since all the services can potentially run as separate processes, an Ariel agent would need to find each of its running services on startup and "introduce" them to one another (by giving object references for each service to the other services for the same agent). We also needed to manage several startup and shutdown dependencies as well as the services' connections to components outside the Ariel agent. To do so, we developed the IDL interface below, which each of the Ariel services implemented. The core Ariel process would find (via the CORBA Name Service) an object reference for each of the services then use these functions to pass those object references to the other services.

```
interface Admin {
 void connectSMS( in SMS::InternalAgentSMS ref );
 void connectUIS( in UIS::InternalAgentUIS ref );
 void connectTSS( in TSS::InternalAgentTSS ref );
 void connectLS( in LS::InternalAgentLS ref );
 void connectNS( in NS::InternalAgentNS ref );
 void disconnectSMS();
 void disconnectUIS();
 void disconnectTSS();
 void disconnectLS();
 void disconnectNS();

 void init ( in DCI::CrewState initial_state,
             in string xml_init_string );

 void resetExternalConnections();
 oneway void shutdown();
};
```

Only some of the services (e.g., not including the CAS) that we defined for Ariel were contained in the original prototype, and this set is rigidly encoded in the service interface. This rigid interface worked well for the initial prototype for several reasons. First, most of the software development team was just learning CORBA, so the simplicity of the strongly-typed interface made it a more reasonable initial step than something with more levels of indirection requiring run-time casting of object types. Second, it gave us the chance to explore how the DCI application might work (which was previously uncharted territory) without spending too much time on designing plug-and-play connectivity. This allowed us to move quickly to a point where we could test our services together as a whole, and allowed us to make progress on the individual services sooner than if we had spent more time on perfecting the connectivity. In doing so, we also learned or confirmed what kind of improvements we needed in the connectivity design for future generations of the implementation.

Beyond the Prototype - Desires

As soon as we demonstrated the initial prototype in the life support domain, we began to want more out of the system than the initial design supported. These desires were driven by several factors:

- **Applications.** The Ariel prototype used a fixed set of services for a particular application supporting NASA astronauts interacting in specific ways with an autonomous control agent for life support. However, we wanted to be able to add the Command and Authorization Service (CAS), which we had not been able to include in the prototype. This would have required rewriting the Admin interface above to include the CAS and changing the implementation of every existing service to accommodate this interface change. In addition, we were asked to apply the Ariel agent in a demonstration to provide notification for ground personnel supporting space station operations. This required removing the CAS and the Task Status Service from the agent. Accommodating this reduced set of services would have also required rewriting the interface (in a different way) and tracking this change in the implementations of all the services.
- **Testbed**. Customizing the service sets is a recurring need for the DCI system. In addition to providing a useful monitoring and control capability as implemented, the DCI environment could also provide a testbed for evaluating new or alternate services from external sources such as universities and research institutes. This would potentially require disabling existing services as well as integrating with new services.
- **Robustness.** The DCI prototype was used for several one-shot demonstrations, but the mature system needed to support continuous 24/7 operation over months. Robustness and graceful degradation in the face of software failures are critical. If an error occurs in one service, the others should be able to carry on without failing until the broken service is replaced. A failure in one service should not cause the entire system to need to be restarted.
- **Efficiency.** Running each Ariel service in a separate process was an excellent strategy, initially, to support distributed development and debugging. However, once the focus shifted to operations rather than building the system, we needed to make the system more lightweight.
- **Convenience.** The coordinated startup and shutdown of ~40 machine processes in the initial prototype was tedious and error prone. We wanted to streamline the

startup of these processes and coordinate the creation of client and server objects in the software where possible. We also wanted to add the ability to recover from user errors in startup order, should they occur.

Beyond the Prototype - Requirements

Based on these desires, we developed a set of requirements for the next generation of the DCI system. These requirements are listed here in order of importance:

1. Make the set of ARIEL services be non-rigid and able to be specified at startup/run-time. Eliminate compile time dependencies among services completely. Eliminate run-time dependencies or include exception handling for "graceful degradation" if other services are not present at run-time. Because different services are responsible for maintaining different (possibly new) parts of the user's state, eliminate the rigid, monolithic "CrewState" data structure, which we had used to hold the entire model of the user's state (originally a NASA crew member). (The user's state may include his or her current activity, location, health, role, etc.) Instead, allow different services to post and read the relevant parts of the agent's beliefs from the SMS, as needed.
2. Make the running ARIEL service set be dynamically re-configurable for robustness at run-time and for easier maintenance and debug. Allow individual services to be disconnected, shutdown, restarted, reinitialized, and reconnected without shutting down the agent or the other services.
3. Streamline start-up and management of the ARIEL agent and services. Where possible, implement the ability to spawn all services from a single initial thread.
4. Make ARIEL agents (along with other DCI components) more lightweight by providing capabilities to combine services in same JVM/Lisp machine processes.

Beyond the Prototype - Design

To meet the first requirement and make the set of services Ariel supports completely extensible, we needed to redesign the definition of a basic Ariel service to eliminate references to specific service types. We followed a plug-and-play paradigm, and used a backbone, called the Ariel Service Manager, that allowed any number of services (0 to many) to be connected. In the new design, each service connected to the others using an untyped object reference and a name. (See IDL definition below.) If one service recognizes the name of another service being connected, it can attempt to resolve the provided object reference and initiate type-specific interactions with that service. Run-time checks for the existence of a service must be in place before one service can call another. We used an XML configuration file to specify the number of services and their types for each Ariel at startup, and we added more structure to the startup process than had previously existed (so that, for example, each service would know when it had received all of the other service references that were going to be available for this instance of the agent). We reimplemented belief access through the SMS as a blackboard where each service could post and read the beliefs it was concerned about without (1) being burdened by state information it did not need in the previous rigid user state data structure and (2) without having to track implementation changes every time a new type of state information was added to the state data structure. Finally, each service was responsible for operating in degraded mode if state information it normally relied upon was not available in this instance of the agent (i.e., if the service acting as the source of that state information failed or was not included in the configuration for this particular agent instance). For example, the Task Status Service can normally use location information to help determine which task a person is currently performing. However, if location state information is not available in this instance of the agent (perhaps the Location Service is not used because location-tracking technology is not installed where the agent will be deployed), then the Task Status Service should be able to make an estimate of the user's current task without the location information and without crashing because the location information is not available.

As a complement to the capability of being able to configure an Ariel agent for any number of services, the design to address the second requirement allowed us to dynamically start, stop, initialize, and connect services as the agent was running. To meet this requirement, we needed to support asynchronous interaction among the services during normal operation of the agent, but also bring the services into synchronized startup and shutdown sequences to allow dynamic service replacement, shutdown, or startup.

```
enum ServiceState { STARTING, RECEIVING_SERVICE_-
REFS, CONNECTING_SERVICES, READY_TO_RUN_OR_-
RECONFIG, RETRACTING_SERVICE_REFS,
DISCONNECTING_SERVICES, STOPPING, RUNNING };

interface Admin-
{
 void enterState( in ServiceState state_id );
 oneway void activateCurrentState(
                in AdminStateCallback ref );
 void stopCurrentState( );

 void setUniqueId( in string service_id );
 void init ( in string xml_init_string );
 void connectService( in string service_name,
                      in Object ref );
 void disconnectService(in string service_name);
 void resetExternalConnections();
 oneway void kill();
};
```

Extending previous work on coordinating multiple agent submodules using state machines (McKay, 1999), we defined a simple state machine for startup and shutdown sequences as well as a RUNNING state in which

- Note: All IDL definitions have been edited for brevity.

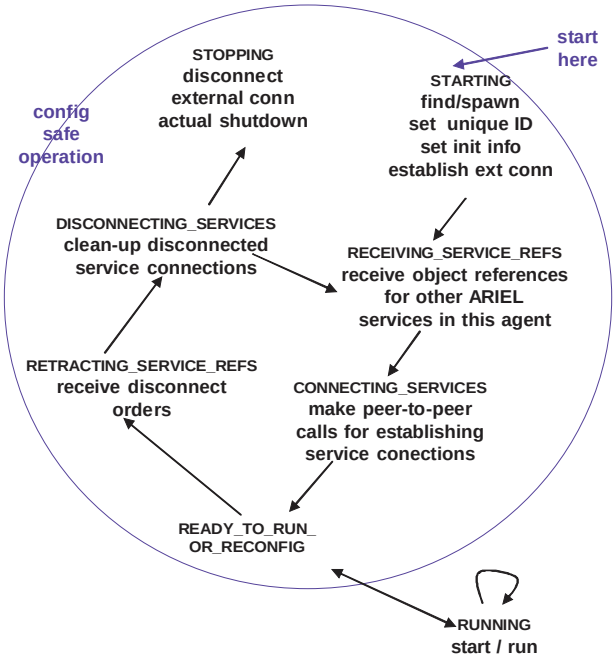

Figure 2. Dynamic Reconfiguration States

asynchronous interactions among the services were allowed (see Figure 2). The IDL definition, above, enumerates each of the possible states. In the new design, this Admin interface replaces the previously presented IDL definition of the Admin interface and is implemented by each service. Three of the functions in this interface move a service through the states. These transitions are synchronized among all the services by the Ariel Service Manager backbone of the Ariel agent. Figure 2 describes which of the other functions are called in each state. When it is time to shutdown a service, all services move from the RUNNING state through the cycle of service connections (only those services being shutdown must be disconnected). A service being shutdown exits the cycle at the STOPPING state. It can be replaced by a new service in the STARTING state, which then joins the existing services to complete the cycle back to the RUNNING state. This cycle can be repeated as often as needed.

To address the last two requirements, we implemented the ability for the Ariel agent to spawn services in addition to looking them up on the CORBA Name Service (which requires them to be started up independently before the Ariel agent goes to find them). For Java components, we are able to spawn services using direct object instantiation inside the same process running the Ariel backbone. We are also able to spawn objects remotely using object factories. A factory is a design pattern for instantiating objects on request. In this case, a software process contacts a factory running on a remote machine, asks it to instantiate an object of a particular type on that machine, and the factory returns a reference to that object once it has been instantiated.

Related Work

We are not aware of any previous descriptions of the software engineering approach for the design and development of a deployed personal-assistant agent system. However, in this section, we discuss several projects that are related to this work along one or more dimensions.

First, the Electric Elves project provides user proxy agents, called "Friday" agents, to every person in an organization (deployed for the research organization developing the Electric Elves project). These Friday agents use AI-based technology to perform the following organizational tasks (Chalupsky et al., 2001; Scerri et al., 2001): (1) monitoring the location of each user and keeping other users in the organization informed, (2) rescheduling meetings if one or more users is absent or unable to arrive in time, (3) selecting a member of the organization to give a research presentation at each research meeting, (4) ordering lunch for meetings or individuals, (5) selecting teams of researchers for giving a demonstration out of town and planning their travel, and (6) arranging meetings with visitors based on research interests. The Elves project was a multi-language multi-platform implementation (Chalupsky et al., 2001). For the implementation and deployment of this project, these researchers would have faced many of the same software engineering challenges described in this paper. However, we are not aware of any publications directly describing their software engineering approach at this level. For integration of AI-technologies across agents, they used an agent communication language (KQML) implementation based on the CoABS Grid* communication infrastructure (Chalupsky et al., 2001).

There are multiple publications available describing the software engineering approach for the Sensible Agent Testbed infrastructure (Barber et al., 2000; Barber et al., 2001). Although this is an experimental testbed, rather than a deployed system, and has no relationship to personal-assistant agents, the issues related to the underlying agent architecture are similar to our system. Both agent architectures have submodules/services, and we were able to build on the state machine approach in the Sensible Agent architecture (McKay, 1999).

Another highly successful example of a documented agent infrastructure supporting dynamic reconfiguration is RETSINA. RETSINA is an implemented multi-agent system infrastructure that has been developed for several years at CMU and applied in many domains ranging from financial portfolio management to logistic planning (Sycara et al., 2002). RETSINA allows multiple agents to come and go in an open system using dynamic matchmaking to allow agents to find other agents with capabilities they need. However, the primary work focuses on communication and interaction among the agents as peers

* http://coabs.globalinfotek.com/public/Grid/What_is_Grid.htm

in a multi-agent system. In contrast, the design focus for the Ariel agents in DCI is the flexible structure of the agents themselves, which can be configured for a given deployment with the specific set of services needed for a given user in that case.

Conclusions

DCI supports human interaction with complex, mostly-autonomous software control agents. This paper describes the software engineering process of integrating several AI-based services into an agent implementation that provides this support. In particular, we have described requirements and design changes needed to achieve a workable prototype with a reasonable amount of effort and then move the system from a prototype to a system ready to be applied to multiple types of real world problems.
The following list summarizes our lessons learned for making such a system extensible, robust, and easy to use:

- Encapsulate the complexity of AI-based services behind well-defined, data-based communication interfaces to reduce integration complexity and provide an upgrade path that does not require implementation changes outside the service.
- Use run-time typing for object references to isolate implementation changes needed when adding new modules or services. Only those components whose internal operation is affected should need to be changed.
- Avoid compile-time dependencies among services, and be diligent about error checking and handling for any run-time dependencies that might exist, including data dependencies. Each service should be able to function (at least to avoid crashing) on its own.
- Isolate each service from run-time failures in other services. Include run-time error checking. Allow healthy services to continue running while a failed service is shutdown and replaced.
- Implement data transfers as push/pull on demand actions with run-time data typing to support extensibility.
- Streamline startup and shutdown processes to avoid user error and frustration.

These lessons can be summarized as "how to make plug-and-play AI-based services work robustly in a large-scale distributed system providing customized user support." Examined after the fact, these lessons are not surprising, and most trace back to the general object-oriented principles of encapsulation and polymorphism as well as the well-recognized need for run-time exception handling. Although we started with those general principles always in mind, we could not predict, at the beginning, how, where, and when to apply these principles. For example, we found that our bootstrapping period of compile-time typing for the prototype was extremely useful in providing early integration opportunities, refining our design, and making initial progress on the individual AI algorithms running as a part of the whole system. Therefore, this paper provides not only a case study about how to apply general software engineering principles to customizable AI-based services; it also provides an example of a successful roadmap for putting these principles and lessons learned into practice.

References

Barber, K. S., Lam, D. N., Martin, C. E., and McKay, R. M. 2000. Sensible Agent Testbed Infrastructure for Experimentation. In *Proceedings of Workshop on Infrastructure for Scalable Multi-agent Systems at The Fourth International Conference on Autonomous Agents (Agents-2000)*, 17-22. Barcelona, Spain.

Barber, K. S., MacMahon, M. T., McKay, R. M., et al. 2001. An Agent Infrastructure Implementation for Leveraging and Collaboration in Operational and Experimental Environments. In *Proceedings of Second Workshop on Infrastructure for Agents, MAS, and Scalable MAS at the Fifth International Conference on Autonomous Agents (Agents-2001)*, 41-52. Montreal, QC, Canada.

Chalupsky, H., Gil, Y., Knoblock, C. A., et al. 2001. Electric Elves: Applying Agent Technology to Support Human Organizations. In *Proceedings of Innovative Applications of Artificial Intelligence*, 51-58. Seattle, WA.

Elsaesser, C. and Sanborn, J. 1990. An Architecture for Adversarial Planning. *IEEE Transactions on Systems, Man, and Cybernetics* 20(1): 186-194.

Martin, C. E., Schreckenghost, D., Bonasso, R. P., Kortenkamp, D., Milam, T., and Thronesbery, C. 2003. An Environment for Distributed Collaboration Among Humans and Software Agents. In *Proceedings of 2nd International Conference on Autonomous Agents and Multi-Agent Systems*, 1062-1063. Melbourne, Australia.

McKay, R. M. 1999. Communication Services for Sensible Agents. Master's Thesis, Electrical and Computer Engineering, University of Texas at Austin.

Scerri, P., Pynadath, D. V., and Tambe, M. 2001. Adjustable Autonomy in Real-World Multi-Agent Environments. In *Proceedings of Autonomous Agents*, 300-307. Montreal, Canada: ACM Press.

Schreckenghost, D., Martin, C. E., Bonasso, R. P., Hudson, M. B., Milam, T., and Thronesbery, C. 2004. Distributed Operations among Human-Agent Teams for Testing Crew Water Recovery Systems. In *Proceedings of AAAI Workshop on Fielding Applications of Artificial Intelligence*. San Jose, CA.

Schreckenghost, D., Thronesbery, C., Bonasso, R. P., Kortenkamp, D., and Martin, C. E. 2002. Intelligent Control of Life Support for Space Missions. *IEEE Intelligent Systems* 17 (5): 24-31.

Sycara, K., Paolucci, M., van Velsen, M., and Giampapa, J. 2002. The RETSINA MAS Infrastructure. *Autonomous Agents and Multi-Agent Systems*.